# A Deterministic Multi-Way Rendezvous Library for Haskell

Nalini Vasudevan
Department of Computer Science
Columbia University
New York, USA
naliniv@cs.columbia.edu

Satnam Singh
Microsoft Research Cambridge
Cambridge CB3 0FB
United Kingdom
satnams@microsoft.com

Stephen A. Edwards
Department of Computer Science
Columbia University
New York, USA
sedwards@cs.columbia.edu

## Abstract

*The advent of multicore processors requires mainstream concurrent programming languages with high level concurrency constructs and effective debugging techniques. Unfortunately, many concurrent programming languages are non-deterministic and allow data races.*

*We present a deterministic concurrent communication library for an existing multi-threaded language. We implemented the SHIM communication model in the Haskell functional language, which supports asynchronous communication and transactional memory. The SHIM model uses multi-way rendezvous to guarantee determinism. We describe two implementations of the model in Haskell, demonstrating the ease of writing such a library.*

*We illustrate our library with examples and experimentally compare two implementations. We also compare our new model with equivalent sequential programs and parallel versions using Haskell's existing concurrency mechanisms.*

## 1. Introduction

Multicore shared-memory multiprocessors now rule the server market. While such architectures provide better performance per watt, they present many challenges.

Scheduling—instruction ordering—is the biggest issue in programming shared-memory multiprocessors. While uniprocessors go to extremes to provide a sequential execution model despite caches, pipelines, and out-of-order dispatch units, multiprocessors typically only provide such a guarantee for each core in isolation; instructions are at best partially ordered across core boundaries.

Controlling the scheduling on multiprocessors is crucial not only for performance, but because data races can cause scheduling choices to change a program's function. Worse, the operating system schedules non-deterministically.

Non-deterministic functional behavior arising from timing variability—a data race—is among the nastiest thing a programmer may confront. It makes debugging all but impossible because unwanted behavior is rarely reproducible [1]. Re-running a non-deterministic program on the same input usually does not produce the same behavior. Inserting assert or print statements or running the program in a debugger usually changes timing enough to make the bug disappear. Debugging such programs is like trying to catch a butterfly that is only visible from the corner of your eye.

We believe a programming environment should always provide functional determinism because it is highly desirable and is very difficult to check for on a per-program basis. Sequential programming environments for languages such as assembly, C, or BASIC have always guaranteed such determinism, but many parallel environments do not.

In this paper, we present a concurrency library that guarantees deterministic functional behavior. We implemented this library in the functional language Haskell, but many of the lessons apply to any language. Haskell actually supports several concurrency mechanisms, but does not guarantee functional determinism. We chose Haskell because it has a fairly mature STM implementation, carefully controlled side effects, and lightweight user-mode scheduled threads. We were also curious about whether our model, which we proposed previously for an imperative setting, would translate well to a functional language.

Our library is based on Edwards and Tardieu's SHIM (software-hardware integration medium) model [5], which consists of concurrent asynchronous processes that communicate exclusively through rendezvous communication channels. Processes only interact at communication points, where they rendezvous and synchronize. The model combines the functional determinism of Kahn networks [9] with the more tractable rendezvous of Hoare's CSP [7]. Our library provides the multi-way rendezvous, but not the exceptions, of the latest version of SHIM [12].

Deadlock is another bane of parallel programming. While the SHIM model does not prevent deadlocks, they are at least deterministic, i.e., if a program can deadlock for a particular input, it will do so consistently. Again, this aids debugging—a programmer can reliably test whether a deadlock has been eliminated for a particular input.

We implemented two versions of our library: one that uses mailboxes for inter-process communication and use that uses software transactional memory. Experimentally, we find mailboxes are more efficient for implementing the multi-way rendezvous mechanism, especially for large numbers of processes. We also found our library easier to code using mailboxes. While our results are most relevant to implementing rendezvous, they suggest not all styles of concurrent communication are equal.

After reviewing the SHIM communication model, some related work, and Haskell's concurrency model, we describe our library and its implementation in Section 5 and present a series of experiments with our library on an eight-processor machine in Section 6.

## 2. The SHIM Communication Model

The SHIM model guarantees functional determinism by restricting inter-thread communication to a multi-way rendezvous mechanism. It has been implemented in two languages [5, 12]. One goal in this paper is to examine the advantages and disadvantages of implementing this model in a library instead of a full-blown language.

Unlike Haskell, the SHIM language [12] is imperative with a C-like syntax. SHIM provides function calls with pass-by-value and pass-by-reference parameters, but no pointers, references, or support for recursive data structures. Its syntax and semantics allow the compiler to perform an inexpensive syntactic check for possible data races. Basically, if no variable is passed by reference more than once at a parallel call site, the program is deterministic. This is a brute-force way to guarantee unique ownership of each variable; other researchers have proposed richer mechanisms such as ownership and region types.

In our library, we adopt two SHIM language constructs: *p par q* runs statements *p* and *q* in parallel, waiting for both to terminate before proceeding; *next c* is a blocking communication operator that synchronizes on channel *c* and either sends or receives data depending on which side of an assignment (=) the *next* appears. The SHIM language also provides concurrent exceptions [12]; our library currently does not implement them.

```
void f(int a) {
  // a is a copy of c
  a = 3; // change local copy
  next a; // receive (wait for g)
  // a now 5
}

void g(int &b) { // b is an alias of c
  next b = 5; // sets c and send (wait for f)
  // b now 5
}

void main() {
  int c = 0;
  f(c); par g(c); // start f() and g() concurrently
}
```

**Figure 1. A SHIM program**

In Figure 1, functions *f* and *g* run in parallel and communicate on channel *c*. Channel *c* is passed by value to *f*, and SHIM interprets the *next* in *f* as a receive since it is an rvalue. Channel *c* is passed by reference to *g* and its *next* is a send because it is an rvalue. The *next* in *f* waits for *g* to send a value. The functions therefore rendezvous at their *next*s, then continue to run after communication takes place.

Inter-process communication in SHIM requires synchronization, and is thus more costly than intra-process communication (i.e., reading and writing local variables). Its implementation on a multiprocessor usually involves locking a shared data structures. As usual, effective algorithms must balance local computation with communication.

### 2.1. SHIM as a Library Versus a Language

The SHIM model provides functional determinacy irrespective of being implemented as a language or a library, so an obvious question is which is preferred. We present the library approach in this paper. A library can leverage existing language facilities (editors, compilers, etc.) but does not provide guarantees about its misuse. A program that uses our library is functionally deterministic if it only uses our library for inter-thread communication, but there is nothing to prevent other mechanisms from being used.

The SHIM language does not provide any other inter-thread communication mechanism, guaranteeing determinism. However, the SHIM language and compiler are not as mature or feature-rich as Haskell, the implementation vehicle for our library.

## 3. Related Work

The advent of mainstream multicore processes has emphasized the challenges of concurrent programming. Techniques ranging ranging from new concurrent languages to new concurrent libraries for existing languages are being investigated. C$\omega$ [2] is an example of a new research language, which provides join patterns in the form of chords that synchronize the arrival of data on multiple channels to atomically capture and bind values that are used by a handler function (such chords are also easy to implement in an STM setting). This pattern can capture many kinds of concurrency mechanisms, including rendezvous and actors, but it is non-deterministic and suffers from all the debugging challenges the SHIM model avoids.

Cilk [3] is another C-based language designed for multithreaded parallel programming that exploits asynchronous parallelism. It provides deterministic constructs to the programmer, but it is the programmer's responsibility to use them properly; the compiler does not guarantee determinism. This is one of the major differences between SHIM and Cilk. Cilk focuses on the runtime system, which estimates the complexities of program parts.

We built our library in Haskell, a functional language with support for concurrency [8]. Its concurrency mechanisms are not deterministic; our library provides a deterministic layer over them. Experimentally, we find such layering does not impose a significant performance penalty.

Our library resembles that of Scholz [11], which also provides an existing concurrency model in Haskell. Unlike Scholz, however, we implement our mechanisms atop the existing concurrency facilities in Haskell [8] and insist on functional determinism.

## 4. Concurrency in Haskell

We built our deterministic communication library atop Haskell's concurrency primitives. The most basic is *forkIO*, which creates an explicit thread and does not wait for its evaluation to complete before proceeding.

We implemented two versions of our library: one using mailboxes [8] for inter-thread communication, the other using software transactional memory [6, 4]. On a mailbox, *takeMVar* and *readMVar* perform destructive and non-destructive reads; *putMVar* performs a blocking write. Similarly, within the scope of an *atomically* statement, *readTVar* and *writeTVar* read and write transactional variables. Other threads always perceive the actions within an *atomically* block as executing atomically.

The Haskell code in Figure 2 creates a mailbox `m` and forks two threads. The first thread puts the value 5 into `m` and the second thread takes the value from the mailbox `m`, adds one to it, and puts it in mailbox `n`.

```
sampleMailbox
 = do
  m <- newEmptyMVar  -- Create a new mailbox
  n <- newEmptyMVar
  forkIO (putMVar m (5::Int))  -- thread writes 5 to m
  forkIO (do
            c <- takeMVar m  -- thread reads m
            putMVar n (c+1))  -- write to n
  result <- takeMVar n  -- block for result
  return result
```

**Figure 2. Using Mailboxes in Haskell. One thread writes to mailbox *m*, a second reads *m*, adds one, and writes to mailbox *n*. The outer thread blocks on *n* to read the result.**

```
sampleSTM c
 = atomically (do
     value <- readTVar c
     if value == -1 then
         retry  -- not written yet
       else writeTVar c (value + 1))
```

**Figure 3. A Haskell program using STM. This updates the shared ("transactional") variable *c* when it is not −1, otherwise blocks on *c*.**

Haskell's software transactional memory mechanisms [6, 4] are another way to manage communication among concurrent threads. In STM, threads can communicate or manipulate shared variables by reading or writing transactional variables. Statements within an *atomically* block are guaranteed to run atomically with respect to all other concurrent threads. A transaction can block on a *retry* statement. The transaction is rerun when one of the transaction variables changes.

The code in Figure 3 reads $c$ and updates it if its value is not $-1$. The *atomically* guarantees the read and write appear atomic to other threads. The thread blocks while $c$ is $-1$, meaning no other thread has written to it.

## 5. Our Concurrency Library

In this section, we present our SHIM-like concurrency library and its implementation. Our goal is to provide an efficient high-level abstraction for coding parallel algorithms that guarantees functional determinism. As described above, Haskell already has a variety of concurrency primitives (mailboxes and STM), but none guarantee determinism. Our hypothesis is that determinism can be provided in an efficient, easy-to-code way.

```
produce [c]
 = do
      val <- produceData
      dSend c val
      if val == -1 then -- End of data
           return ()
        else
           produce [c]

consume [c]
 = do
      val <- dRecv c
      if val == -1 then -- End of data
          return ()
        else
          do consumeData val
             consume [c]

producerConsumer
 = do
     c <- newChannel
     (_,_) <- dPar produce [c]
                   consume [c]
     return ()
```

**Figure 4. A simple producer-consumer system using our library**

### 5.1. Our Library's API

Our library provides channels with multi-way rendezvous and a facility for spawning concurrent threads that communicate among themselves through channels.

Figure 4 illustrates the use of our API. The *producerConsumer* function uses *newChannel* to create a new channel *c* and passes it to the *produce* and *consume* functions, which *dPar* runs in parallel. The producer sends data to the consumer, which consumes it while the producer is computing the next iteration. For communication costs not to dominate, evaluating *produceData* and *consumeData* should be relatively costly. Depending on which runs first, either the *dSend* of the producer waits for *dRecv* of the consumer or vice-versa, after which point both proceed with their execution to the next iteration.

Such a mechanism is also convenient for pipelines, such as Figure 5. The four functions run in parallel. The first feeds data to *pipelineStage1*, which receives it as *val1*, processes it and sends the processed data *val2* to *pipelineStage2* through channel *c2*. *PipelineStage2* acts similarly, sending its output to *outputFromPipeline* through *c3*.

Figure 6 shows the formal interface to our library. *newChannel* creates a new rendezvous channel. *dPar* takes four arguments: the first two are the first function to run and the list of channels passed to it; the last two are the second

```
inputToPipeline [c1]
 = do
      val1 <- getVal
      dSend c1 val1
      inputToPipeline [c1]

pipelineStage1 [c1, c2]
 = do
      val1 <- dRecv c1
      val2 <- process1 val1
      dSend c2 val2
      pipelineStage1 [c1, c2]

pipelineStage2 [c2, c3]
 = do
     val2 <- dRecv c2
     val3 <- process2 val2
     dSend c3 val3
     pipelineStage2 [c2, c3]

outputFromPipeline [c3]
 = do
     val3 <- dRecv c3
     putStrLn (show val3)
     outputFromPipeline [c3]

pipelineMain
 = do
     c1 <- newChannel
     c2 <- newChannel
     c3 <- newChannel
     let dPar2 fun1 clist1 fun2 clist2 clist
           = dPar fun1 clist1 fun2 clist2
     let forkFunc1 = dPar2 inputToPipeline [c1]
                           pipelineStage1 [c1, c2]
     let forkFunc2 = dPar2 pipelineStage2 [c2,c3]
                           outputFromPipeline [c3]
     dPar forkFunc1 [c1, c2]
          forkFunc2 [c2, c3]
     return ()
```

**Figure 5. A two-stage pipeline in our library**

function and its channel connections. *dSend* takes two parameters: the channel and the value to be communicated. *dRecv* takes the channel as argument and returns the value in the channel.

#### 5.1.1 Deadlocks and Other Problems

While our library guarantees functional determinism, it does not prevent deadlocks. For example, our library deadlocks when multiple threads call *dSend* on the same channel (a channel may only have one writer). While this could be detected, other deadlocks are more difficult to detect. If no sender ever rendezvous, the readers will block indefinitely.

```
newChannel :: IO (Channel a)
dPar :: ([Channel a] -> IO b) ->
        [Channel a] ->
        ([Channel a] -> IO c) ->
        [Channel a] -> IO (b,c)
dSend :: Channel a -> a -> IO ()
dRecv :: Channel a -> IO a
```

**Figure 6. The interface to our concurrency library. *newChannel* creates a new channel; *dPar* forks two threads and waits for them to terminate; *dSend* rendezvous on a channel and sends a value; and *dRecv* rendezvous and receives a value.**

```
data Channel a = Channel {
      connections :: TVar Int,
      waitingReaders :: TVar Int,
      written :: TVar Bool,
      allReadsDone :: TVar Bool,
      val :: TVar (Maybe a)
    }
```

**Figure 7. The channel type (STM)**

Two threads that attempt to communicate on shared channels in different orders will deadlock. For example,

| | |
|---|---|
| *dSend c1 value* | *dSend c2 value* |
| *dRecv c2* | *dRecv c1* |

will deadlock because the left thread is waiting for the right to rendezvous on *c1*, while the right is waiting for the left to rendezvous on *c2*. Such a deadlock is deterministic: the scheduler cannot make it disappear.

## 5.2. An STM Implementation

One implementation of our library uses Haskell's facilities for Software Transactional Memory (STM) [6]. Our goal was to see how difficult it would be to code and how efficient it would be for multi-way rendezvous. We describe the implementation below and defer experimental results to Section 6.

Figure 7 shows the collection of transactional variables used to represent a channel. The type variable *a* makes it polymorphic, *connections* tracks the number of threads that must rendezvous to perform the communication (it is adjusted by threads starting and terminating), *val* holds the data being communicated, *waitingReaders* tracks the number of threads that have blocked trying to read from the channel, *written* indicates whether the writer has written the data, and *allReadsDone* indicates when the last blocked reader has unblocked itself.

```
newChannel
  = do
      connectionsT <- atomically $ newTVar 1
      waitingReadersT <- atomically $ newTVar 0
      writtenT <- atomically $ newTVar False
      allReadsDoneT <- atomically $ newTVar False
      valT <- atomically $ newTVar Nothing
      return (Channel connectionsT waitingReadersT
                      writtenT allReadsDoneT valT)
```

**Figure 8. Creating a new channel (STM)**

```
dPar func1 v1 func2 v2 = do
  done <- newEmptyMVar
  let common =
        intersectBy
          (\ x y -> (val x) == (val y)) v1 v2
  atomically (do
    apply (\ c -> do
        nt <- readTVar (connections c)
        writeTVar (connections c) (nt + 1)
      ) common)
  forkIO (do
    res <- func1 v1 -- Run func1 in child
    putMVar done res) -- Save result
  res2 <- func2 v2 -- Run func2 in parent
  res1 <- takeMVar done -- Get func1 result
  atomically (do
    apply (\ c -> do
        nt <- readTVar (connections c)
        writeTVar (connections c) (nt - 1)
      ) common)
  return (res1, res2)

apply func [] = return ()
apply func (hd:tl) = do func hd ; apply func tl
```

**Figure 9. Our implementation of dPar**

### 5.2.1 Forking parallel threads

Figure 9 shows our implementation of *dPar* for STM. It creates a new MVar to hold the result from the child thread, then determines which channels are shared (*v1* and *v2* holds their names) and atomically increases their *connections*.

To evaluate the two functions, the parent forks a thread. The child thread evaluates *func2* and then writes the result into the mailbox. Meanwhile, the parent evaluates *func1*, waits for the child to report its result, atomically decreases the connection count on shared channels, and finally returns the results from *func1* and *func2*.

Figure 10 illustrates how *connections* evolves as threads fork and terminate. In Figure 10(a), F0 has spawned F1 and F2, increasing *connections* to 2. In (b), F2 has spawned F3 and F4, increasing *connections* to 3. Finally, in (c), F3 and F4 have terminated, reducing *connections* to 2.

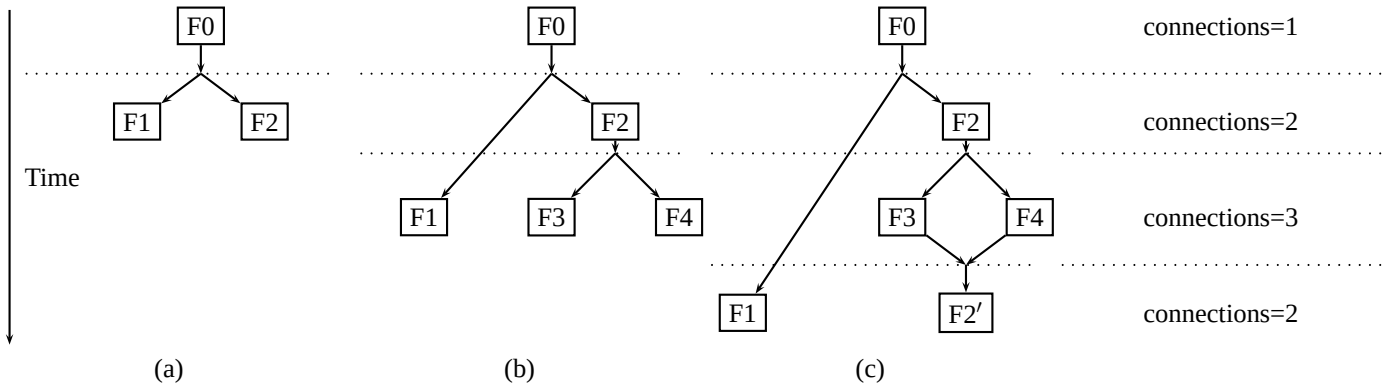


**Figure 10. The effects on *connections* when (a) main function F0 calls *dPar F1 [c] F2 [c]*, then (b) F2 calls *dPar F3 [c] F4 [c]*, and (c) when F3 and F4 terminate.**

Note that this only happens when F0, ..., F4 are all connected to channel *c*. If a thread was not connected, spawning it would not require the number of connections to change. This is what the computation of *common* in Figure 9 accomplishes by looking for channels passed to both threads being started.

### 5.2.2 Deterministic send and receive

Multi-way rendezvous is a three-phase process: wait for all peers to rendezvous, transfer data, and wait for all peers to complete the communication. Our library supports single-writer, multiple-reader channels, so if $n_c$ is the number of threads connected to channel *c*, a writer waits for $n_c - 1$ readers; a reader waits for one writer and $n_c - 2$ other readers. We describe how to maintain $n_c$ in the next section.

Figure 11 illustrates a scenario with two readers and a writer. Threads T1 and T3, call dRecv and dSend respectively. T1 and T3 wait for thread T2 to communicate. Once T2 calls dRecv, the three threads rendezvous and exchange data and continue with their individual execution.

Figure 12 shows our implementation of *dSend* using STM. It first waits for $n_c - 1$ readers to rendezvous, invoking *retry* to delay. Once they have, it atomically writes the value to send in *val* and resets the number of waiting readers, the *written* flag, and the *allReadsDone* flag. Finally, it waits for all the last receiver to set *allReadsDone*.

Figure 13 is the complementary process. It first increments *waitingReaders*, then waits for the *written* flag to be set by *dSend*. Once it has, it reads *val*—the data being communicated, increases *waitingReaders*, and sees if it was the last one. If it was, it resets *waitingReaders*, *allReadsDone*, and *written*, thereby releasing all the readers (including itself) and the writer. Otherwise, it waits for another reader to set *allReadsDone*.

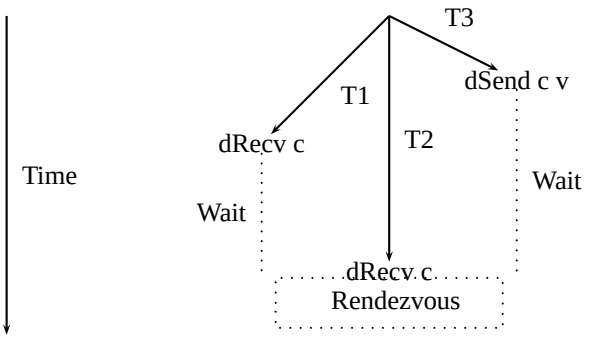


**Figure 11. A rendezvous among two readers and one writer**

```
dSend c value = do
  atomically (do
    wr <- readTVar (waitingReaders c)
    connections <- readTVar (connections c)
    if wr < connections - 1 then retry else (do
      writeTVar (val c) (Just value)
      writeTVar (waitingReaders c) 0
      writeTVar (written c) True
      writeTVar (allReadsDone c) False))
  atomically (do
    ard <- readTVar (allReadsDone c)
    if ard == False then retry else return ())
```

**Figure 12. dSend (STM)**

```
dRecv c = do
  atomically (do
    wr <- readTVar (waitingReaders c)
    writeTVar (waitingReaders c) (wr + 1)
    return ())
  v <- atomically (do
    w <- readTVar (written c)
    if w == False then retry else (do
      Just v <- readTVar (val c)
      wr <- readTVar (waitingReaders c)
      writeTVar (waitingReaders c) (wr + 1)
      nc <- readTVar (connections c)
      -- If last reader to read
      when (wr + 1 == nc - 1) (do
        writeTVar (waitingReaders c) 0
        writeTVar (allReadsDone c) True
        writeTVar (written c) False)
      return v))
  atomically (do
    ard <- readTVar (allReadsDone c)
    if ard == False then retry else return () )
  return v
```

**Figure 13. dRecv (STM)**

```
data Channel a = Channel {
    mVal :: MVar a,
    mVarCount :: MVar Int,
    mVarBegin :: MVar (),
    mVarEnd :: MVar ()
  }
```

**Figure 14. The channel type (Mailboxes)**

### 5.3. A Mailbox Implementation

For comparison, we also implemented our multi-way rendezvous library using Haskell's mailboxes [8].

Figure 14 shows the *Channel* structure used to represent the channel. Field *mVal* holds the data, *mVarCount* holds the number of connections to this channel, and *mVarBegin* and *mVarEnd* are synchronization variables.

Figure 17 shows the *dRecv* procedure. A receiver sends a signal to the sender indicating it has arrived, then the receiver waits for the value from the sender. Once all receivers have read the value, the sender signals an end, after which dRecv returns with the value.

The *dSend* procedure (Figure 16) waits for all receivers, then performs a *putMVar* on the value once per receiver. To ensure the last receiver has read, it does a redundant *putMVar* and *takeMVar*. Finally, once all receivers have read the value, it signals the receivers to continue execution. *WaitForRecvrsToArrive* waits for every receiver to send a sync indicating it has arrived. *SignalRecvrs* signals the end by informing each receiver the rendezvous is complete.

```
newChannel
  = do
      mVal <- newEmptyMVar
      mVarCount <- newMVar 1
      mVarBegin <- newEmptyMVar
      mVarEnd <- newEmptyMVar
      return (Channel mVal mVarCount
                      mVarBegin mVarEnd)
```

**Figure 15. newChannel (Mailboxes)**

```
dSend (Channel mVar mVarCount
               mVarBegin mVarEnd) val = do
  waitForRecvrsToArrive mVarCount mVarBegin 1
  -- Wait for every receiver to send a sync.
  n <- readMVar mVarCount
  sendValueToRecvrs mVar val (n-1)
  putMVar mVar val
  takeMVar mVar
  signalRecvrs mVarEnd (n-1)

sendValueToRecvrs mVar value count = do
    if (count == 0) then
        return ()
      else do putMVar mVar value
              sendValueToRecvrs mVar
                      value (count - 1)
              return ()

waitForRecvrsToArrive mVarCount mVarBegin i
 = do
    count <- readMVar mVarCount
    if (count == i) then return ()
     else do
           takeMVar mVarBegin
           waitForRecvrsToArrive mVarCount
                                 mVarBegin (i+1)

signalRecvrs mVarEnd count
 = do if (count == 0)
       then return ()
      else do putMVar mVarEnd ()
              signalRecvrs mVarEnd (count-1)
```

**Figure 16. dSend (Mailboxes)**

```
dRecv (Channel mVar mVarCount
              mVarBegin mVarEnd)
 = do
    putMVar mVarBegin () -- Inform sender
    value <- takeMVar mVar -- Wait for sender
    takeMVar mVarEnd -- Wait for sender end
    return value
```

**Figure 17. dRecv (Mailboxes)**

| Threads | Time to Rendezvous | | Speedup |
|---|---|---|---|
| | STM | Mailbox | (STM/Mailbox) |
| 2 | 0.11 ms | 0.07 ms | 1.6 |
| 3 | 0.14 | 0.08 | 1.8 |
| 4 | 0.17 | 0.14 | 1.2 |
| 5 | 0.21 | 0.16 | 1.3 |
| 6 | 0.28 | 0.17 | 1.6 |
| 7 | 0.31 | 0.21 | 1.5 |
| 8 | 0.37 | 0.23 | 1.6 |
| 9 | 0.42 | 0.27 | 1.6 |
| 10 | 0.47 | 0.28 | 1.7 |
| 100 | 6.4 | 1.8 | 3.5 |
| 200 | 35 | 6.7 | 5.2 |
| 400 | 110 | 14 | 7.7 |
| 800 | 300 | 34 | 8.9 |

**Table 1. Time to rendezvous for STM and Mailbox implementations**

## 6. Experimental Results

To test the practicality and efficiency of our library, we created a variety of programs that used it and timed them.

### 6.1. STM Versus Mailbox Rendezvous

As a basic test of efficiency, we had our library rendezvous 100 000 times among various numbers of threads on a two-processor machine (a 500 MB, 1.6 GHz Intel Core 2 Duo running Windows XP) and measured the time. Table 1 lists the results.

Mailboxes appear to be more efficient for our application, especially when large numbers of threads rendezvous. We believe this may be fundamental to the STM approach, in which threads continue to execute even if there is a conflict. Only at the end of the transaction is conflict checked and rolled back if needed. In the case of a multi-way rendezvous, many threads will conflict and have to be rolled back. Mailboxes are more efficient here because they check for conflicts earlier.

The STM method also requires more memory to hold the information for a roll-back. Again, mailboxes have less overhead because they do not need this information.

The STM method is more complicated. Unlike mailboxes, which only require a mutual exclusion object, a flag, and the data to be transferred, STM requires managing information to identify conflicts and roll back transactions.

However, the ratio of communication to computation is the most critical aspect of application performance. For a computationally-intensive application, a 50% increase in communication time hardly matters.

```
fib n | n <= 1 = 1
      | otherwise =
          par res1 (pseq res2 (res1 + res2 + 1))
          where res1 = fib (n - 1)
                res2 = fib (n - 2)
```

**Figure 18. Calculating Fibonacci numbers with Haskell's par-seq**

### 6.2. Examples Without Rendezvous

These examples only call *dPar* and do not use *dSend* or *dRecv*. Our goal here is to compare our library with Haskell's existing par-seq facility, which we feel presents an awkward programming interface [10].

Haskell's par-seq constructs can be used to emulate our *dPar*. The following are semantically equivalent

$$dpar\ M\ []\ N\ [] \leftrightarrow (par\ M\ (pseq\ N\ (M, N)))$$

but the *par* does not guarantee *M* and *N* are executed in parallel because Haskell uses lazy evaluation. Nevertheless, we find the par-seq method can run faster than our *dPar*.

Using par-seq is subtle, illustrated by Figure 18. While both *par* and *pseq* only return the value of their second argument, the meaning of *m1 par m2* is "start the calculation of *m1* for speculative evaluation and then go onto evaluate *m2*." This is useful when *m1* is a sub-expression of *m2* so *m1* may be evaluated in parallel with the body of *m2*. Conversely, *pseq* makes sure its first argument is evaluated before evaluating its second. In this example, the *pseq* guarantees that *fib (n-2)* is evaluated before *fib (n-1)*, which can use *fib (n-2)*.

We find this mechanism subtle and difficult to control. It provides weak control over the scheduling of computation—a complex issue for a lazy language like Haskell made all the more tricky by parallelism. We believe providing users with easy-to-use mechanisms to control scheduling is necessary for achieving high performance; expecting the compiler or runtime system to make the best choices seems unrealistic.

We ran these and all later experiments on an 8-processor Intel machine containing two 5300-series 1.6 GHz quad processors, 2 GB of RAM, and running Windows NT Server.

#### 6.2.1 Maximum element in a list

Figure 19 shows the execution times for a program that uses a linear search to find the maximum element in a 400 000-element list. The program, whose behavior is shown in Figure 20, splits a list into pieces, one per thread, finds the maximum of each piece, and finally finds the maximum of

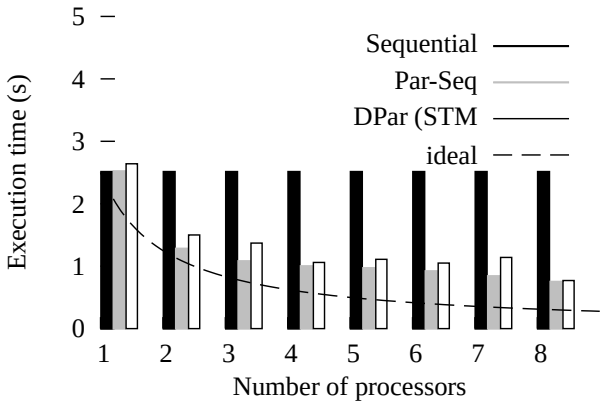


**Figure 19. Maximum Element in a List**

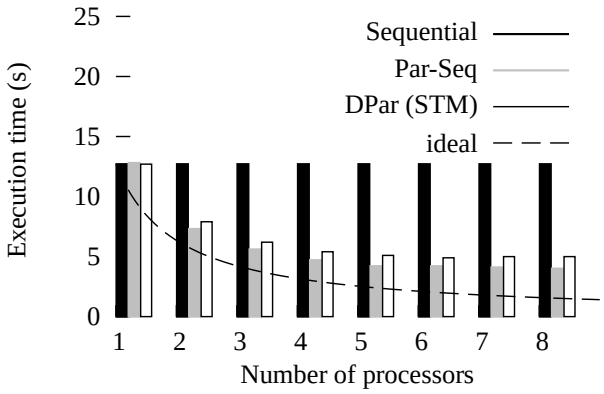


**Figure 21. Boolean Satisfiability**

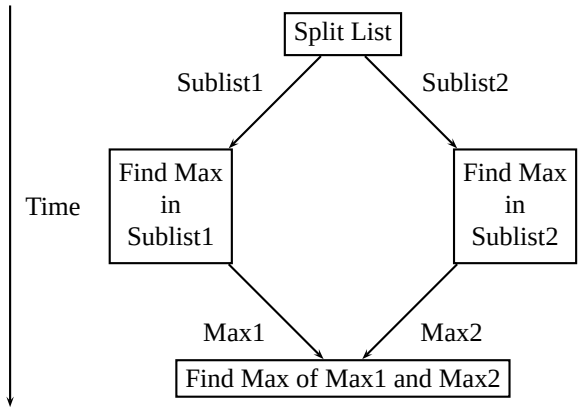


**Figure 20. Structure of Maximum Finder**

the pieces. We compared a sequential implementation, one that uses Haskell's par-seq constructs, and one that uses our *dPar* to the ideal speedup of the sequential implementation.

Figure 19 shows the par-seq implementation is slightly more efficient, although both implementations fall short of the ideal $1/n$ speedup on more than two processors.

### 6.2.2 Boolean Satisfiability

Figure 21 shows the execution times of a simple Boolean SAT solver implemented sequentially, using par-seq, and with our *dPar*. We ran it on an unsatisfiable problem with 600 variables and 2 500 clauses. Figure 22 shows the structure of our approach: we pick an unassigned variable and spawn two threads that check whether the expression can be satisfied if the variable is true or false. Because of our demand for determinism, we do not asynchronously terminate all the threads when one finds the expression has been satisfied. Our algorithm is also primitive in the sense that it does not do any online learning.

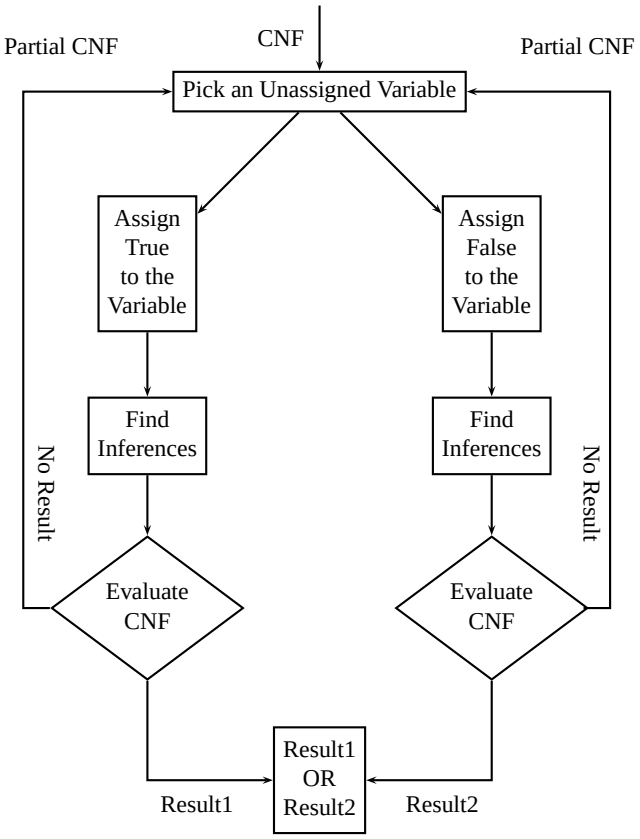


**Figure 22. Structure of the SAT Solver**

Again, we find our *dPar* has more overhead than Haskell's par-seq. Also, this algorithm does not appear to benefit from more than four processors, which we attribute in part to Haskell's single-threaded garbage collector.

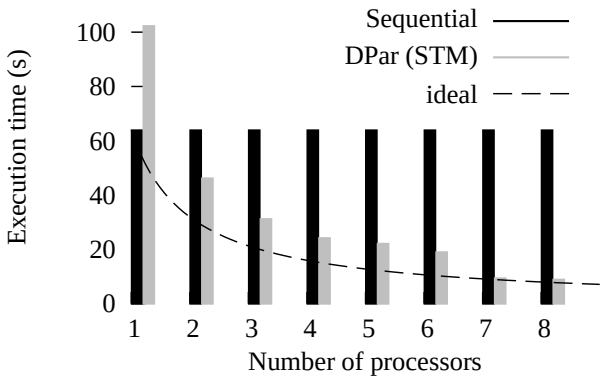


**Figure 23. Times for Linear Search**

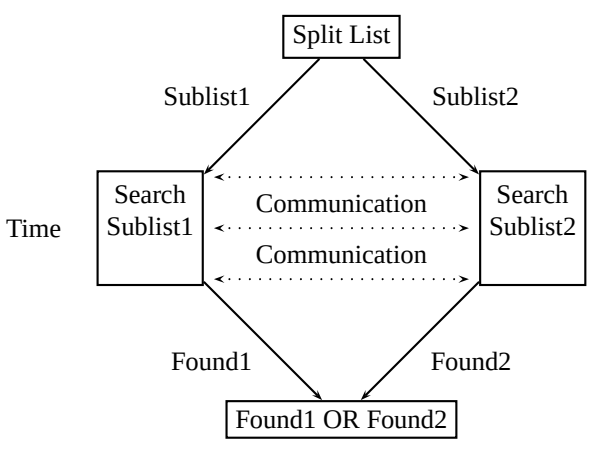


**Figure 24. Linear Search Structure**

### 6.3. Examples With Rendezvous

Here we consider algorithms that use rendezvous communication among threads. The comparisons are to purely sequential implementations of the same algorithm.

#### 6.3.1 Linear Search

Figure 23 shows the execution times of a linear search program that uses rendezvous communication to find a key in a 420 000-element list (we put it in the 390 000th position). Unlike the maximum element problem, linear search generally does not need to scan the list completely, so the algorithm should have a way of terminating early.

Requiring determinism precludes the obvious solution of scanning $n$ list fragments in parallel and terminating immediately when the key is found. This constitutes a race if the key appears more than once, since the relative execution rates of the threads affect which copy was reported.

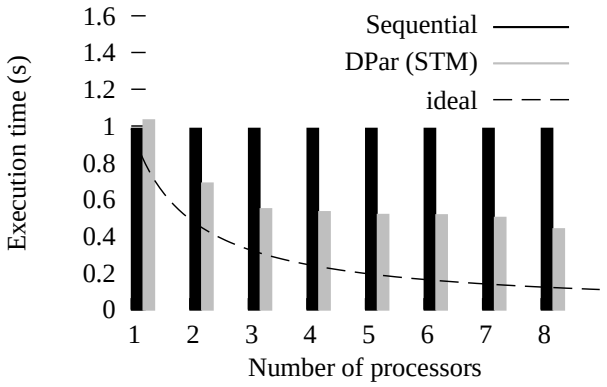


**Figure 25. Systolic Filter**

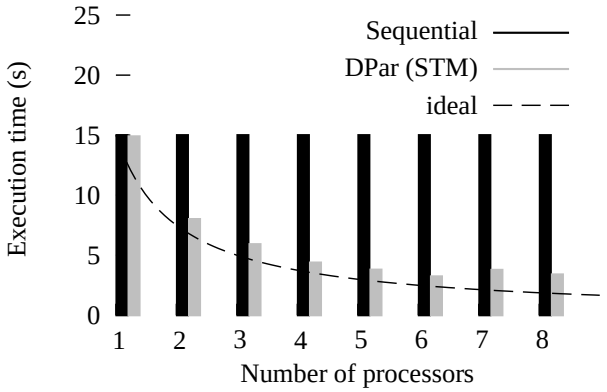


**Figure 26. RGB Histogram**

Our implementation takes the approach shown in Figure 24: the list is broken into $n$ fragments and passed to parallel threads. However, rather than asynchronously terminating all the threads when the key is found, instead all the threads rendezvous at a prearranged interval to check whether any have found the key. All threads proceed if the key is not found or terminate and negotiate which copy is reported if one has been found.

This technique trades off communication frequency and the amount of extra work likely to be done. Infrequent communication means less overhead, but it also makes it more likely the threads will waste time after the key is found. Frequent communication exchanges overhead for punctuality. We did not have time to explore this trade-off.

#### 6.3.2 Systolic Filter and Histogram

Figure 25 shows the execution times of a Systolic 1-D filter running on 50 000 samples. Each thread run by the filter can independently process a chunk of the input in parallel with other threads following the structure in Figure 27. Because

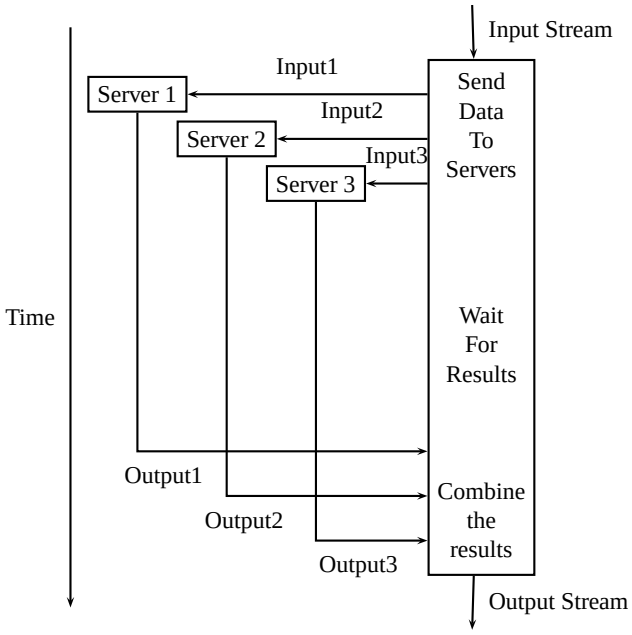


**Figure 27. Server Programming Style used by Histogram and Systolic Filter**

of determinism, jobs are distributed and collected from the worker threads in a round-robin order.

Figure 26 shows the execution time of a similar algorithm: a histogram of RGB values in an image. We ran it on a 565 KB raster file.

## 7. Conclusions

While we found it was reasonable and fairly efficient to implement a deterministic concurrency library based on multi-way rendezvous, our efforts did raise a few issues.

We found that the performance of our library was slightly lower than that of Haskell's built-in par-seq mechanism. We suspect this is from additional layers of abstraction between our library calls and the par-seq mechanism. Despite this, we believe our library provides a nicer abstraction because it makes communication and synchronization explicit and therefore makes an easier optimization target, but this is difficult to quantify.

While we were successful at implementing the library using both Mailboxes and software transactional memory (STM), we are happier with the mailbox-based implementation because it is both faster and easier to program and understand. While it is clearly possible to wait to synchronize with peers in STM, coding it seems needlessly awkward. We also observed STM increased synchronization overhead by at least 50%, although this is not prohibitive.

Our experiences do provide insight for the library vs. language debate. While the library approach has the advantage of leveraging features of the host language, we encountered a number of infelicitous that made the library difficult to implement and use.

Unlike C, Haskell does not allow its type system to be circumvented. This avoids more runtime errors but makes building really polymorphic things harder. We would like a *dPar* that spawns an arbitrary number of threads, each of which is connected to an arbitrary number and type of channels. Such flexibility is difficult to express in a library. We settled on spawning only two threads at a time (*n*-way forks can be recovered by nesting) and not checking the number of channels, thus deferring certain errors to runtime. Haskell probably allows a more flexible interface, but the code can become very obscure.

The type system in C is easy to circumvent and C allows functions with a variable number of parameters, so a C implementation of our library might have a cleaner API. However, going around the type system opens the door to runtime type errors, e.g., trying to pass a string through a channel intended for floating-point numbers.

We believe our library presents an easier, less error-prone programming model than either mailboxes or STM, but this is hard to prove. Anecdotally, we found it easier to debug, especially deadlocks, which were reproducible. Furthermore, it seems easier to reason about explicit synchronization instead of explicitly using *retry* in the STM setting.

Tardieu and Edwards recently added concurrent, deterministic exceptions to the SHIM model [12], which are a convenient mechanism for thread control but tricky to implement correctly. We plan to add such concurrent exceptions to the next version of our library.

## Acknowledgments

We would like to thank Simon Peyton Jones, Simon Marlow and Tim Harris of Microsoft Research for their valuable suggestions and feedback. This research was supported in part by Microsoft Research, Cambridge. Edwards and the SHIM project is supported by the NSF.

## References


[1] Y. Ben-Asher, E. Farchi, and Y. Eytani. Heuristics for finding concurrent bugs. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, page 288, Nice, France, Apr. 2003.

[2] G. M. Bierman, E. Meijer, and W. Schulte. The essence of data access in C$\omega$. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 287–311, Glasgow, UK, July 2005.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[4] A. Discolo, T. Harris, S. Marlow, S. P. Jones, and S. Singh. Lock free data structures using STM in Haskell. In *Proceedings of Functional and Logic Programming (FLOPS)*, volume 3945 of *Lecture Notes in Computer Science*, pages 65–80, Fuji Susono, Japan, Apr. 2006.

[5] S. A. Edwards and O. Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):854–867, Aug. 2006.

[6] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP)*, pages 48–60, Chicago, Illinois, USA, June 2005.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, New Jersey, 1985.

[8] S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 295–308, Florida, USA, Jan. 1996.

[9] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, Aug. 1974. North-Holland.

[10] P. Roe. *Parallel Programming Using Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, Scotland, Feb. 1991.

[11] E. Scholz. Four concurrency primitives for Haskell. In *ACM/IFIP Haskell Workshop*, pages 1–12, La Jolla, California, June 1995. Yale Research Report YALE/DCS/RR–1075.

[12] O. Tardieu and S. A. Edwards. Scheduling-independent threads and exceptions in SHIM. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 142–151, Seoul, Korea, Oct. 2006.